Posts /

Crew AI Crash Course (Step by Step)

28 March 2024 · 15 mins

AUTHOR

Alejandro AO

I'm a software engineer building AI applications. I publish weekly video tutorials where I show you how to build real-world projects. Feel free to visit my YouTube channel or Discord and join the community.

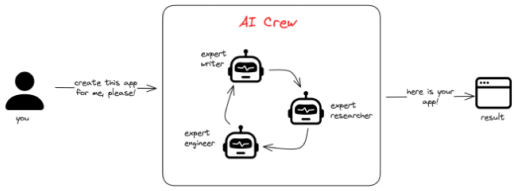
Quick Links

- GitHub Repo
- YouTube Video
- CrewAI
- Support me on Patreon ❤️

Introduction

Hey there. This is an introduction to CrewAI. We will explore everything you need to know to get started with CrewAI, starting from the basics.

> *CrewAI is a new tool that allows us to create our own teams of autonomous agents, all of them working for us.*



Each agent is an expert in a different task, and they all work together to help us achieve our goal. This goal can be either crafting an email based on some research, creating a business plan, writing a book, creating a blog post, etc.

The possibilities are infinite.

All we have to do is think of the process and tasks that would be required to achieve our goal, and assign those tasks to our crew of agents. They will do all the work for us.

In this tutorial, we will introduce you to the basics of CrewAI and show you how to create a simple crew using a sequential process (more about what this means later). This is a completely hands-on tutorial, so make sure you have your development environment set up and ready to go.

Prerequisites

Before we start, make sure you have the following installed:

- A Python environmehnt running at least Python 3.10 (you can use conda envs for this)
- Some basic knowledge of Python
- No prior knowledge of CrewAI is required

Use case example

In this tutorial, we will be focusing on one of the examples published by CrewAI themselves. In my opinion, the prep-for-a-meeting example is one of the best for beginners. It is complex enough to give you a good overview of the power of CrewAI, but without being too complex that a beginner would not understand it.

Define the goal of the crew

Remember that, when building applications using CrewAI, we are essentially creating a team of AI agents that work for us. We need to tell them what we expect from them.

In this case, we want to create a crew that will help us prepare for a meeting. But idealy, you would want to use your crew several times for similar tasks.

In other words, you should be able to reuse your crew anytime you want to prepare for any meeting. So we are going to need to pass in a user input that will make the goal of the crew more specific.

User input

In this case, before kickstarting our crew, we are going to ask the user for the following information:

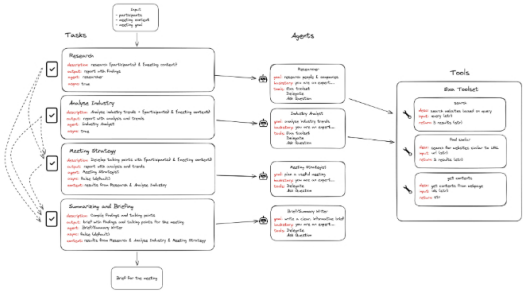
- The names of the participants who are going to be in the meeting.
- The context of the meeting.
- The overall goal of the meeting.

Expected output

In return, our crew should be able to give us a detailed brief for the meeting. This brief should contain relevant information about the industry, the people involved in the meeting, and relevant talking points that will help us get to our the goal we sent as input.

In order to set up this crew, we should consider the following concepts: agents, tools, tasks and processes.

1. **Tasks**: These are the tasks that our agents will perform. Each task will be assigned to an agent.
2. **Agents**: These are the AI agents that will be working for us. Each agent is an expert in a different task. In this case, we will have 4 agents.
3. **Tools**: These are the tools that our agents will use to perform their tasks. These can be, for example, a search engine, a summarizer, a translator, etc.
4. **Process**: A process dictates the way that our agents will work together. In this case, we will use a sequential process, which means that each agent will work one after the other. (More about this later).



As you can see in the diagram above, each task, agent and tool has its own properties.

How to plan your crew

The diagram above shows very well that the main axis when assempling our crew is the list of tasks that our crew will perform. Think if it as a to-do list.

In this case, the main tasks are:

1. Research participants
2. Analyze industry
3. Prepare talking points
4. Summarize brief

Tasks

Let's delve a little deeper into the `Task` object. In CrewAI, each `Task` has the following properties:

- **Description**. For example, "Search the web for relevant information...". Try to make it as detailed as possible. This will be part of the prompt that the agent will use to perform the task.
- **Agent**. The agent that will perform this task.
- **Context**. Other information needed to perform the task. Usually, this is the output from the previous tasks.
- **Async Execution**. Whether or not the task should be executed simultaneously with other tasks. In our example, the first two tasks (`Research participants` and `Analyze industry`) can be executed simultaneously, while the last two tasks (`Prepare talking points` and `Summarize brief`) should not. This is because the last two tasks depend on the output of the first two tasks.

There are other properties that we can set for each task, you can check out the Task class in the CrewAI documentation for more information.

Here is what a `Task` object looks like in CrewAI:

```
def research_task(self, agent, participants, meeting_context):
  return Task(
    description=dedent(f"""\
      Conduct comprehensive research on each of the individuals and companie
      involved in the upcoming meeting. Gather information on recent
      news, achievements, professional background, and any relevant
```

```
      business activities.

      Participants: {participants}
      Meeting Context: {meeting_context}"""),
    expected_output=dedent("""\
      A detailed report summarizing key findings about each participant
      and company, highlighting information that could be relevant for the m
    async_execution=True,
    agent=agent
  )
```

Agents

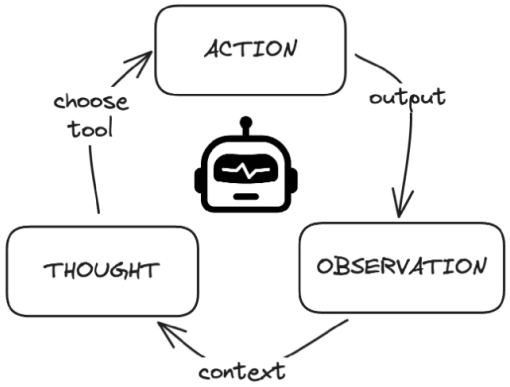
Once that we have our tasks defined, we need to assign them to agents. But what are agents?

You can think of agents as the workers in our crew. They are the ones responsible for performing each task, and each agent is an expert in a different task. Here is what an `Agent` object looks like in CrewAI:

```
def research_agent(self):
  return Agent(
    role='Research Specialist',
    goal='Conduct thorough research on people and companies involved in the
    tools=ExaSearchTool.tools(),
    backstory=dedent("""\
        As a Research Specialist, your mission is to uncover detailed inform
        about the individuals and entities participating in the meeting. You
        will lay the groundwork for strategic meeting preparation."""),
    verbose=True
  )
```

How Agents Think

Agents in CrewAI are essentially ReAct agents coded with LangChain. They are, in a sense, autonomous. This means that they can think for themselves and complete their tasks without us having to tell them what steps to take. All they need is an initial prompt telling them what we expect from them. In this case, we expect them to perform the task that we assigned to them.



In order to do this, they follow the following steps in a loop:

1. **Thought**: Given some context, they use the LLM to "think" what would be the right action to take. For example, after passing the task, they might think "I need to search the web for relevant information about the participants".
2. **Action**: Given the thought that they just had, they use the LLM again to formulate the action that they want to perform. They choose this action out of a defined list of possible actions. Each agent knows which actions they can perform because we declared them when creating the agent.
3. **Action input**: Each action is essentially a function. So we need to pass in some input to this function. For example, the action might be "search the web", so it will take a search query, such as "who is the CEO of Google?".
4. **Observation**: The agent will then perform the action and give us back the result. This result is called an observation. In the case of the "search the web" action, the observation would be the search results.

After the first iteration, the agent will take the observation, append it to the context, and then repeat the loop. If at any point the agent thinks that it has completed the task, it's thought will be something like "I now have all the information that I need to complete the task". At this point, the agent will stop the loop and give us the final result.

How to create an agent

Remember that, when dealing with LLMs, a good prompt engineering technique is to give the LLM a role to play. This is what we are doing when defining their `backstory`. We are creating one agent for each task, and we are giving each agent a specific background so that they can perform their task correctly.

```
def research_agent(self):
  return Agent(
    role='Research Specialist',
    goal='Conduct thorough research on people and companies involved in the
```

```
    tools=ExaSearchTool.tools(), # array with tools
    backstory=dedent("""\
        As a Research Specialist, your mission is to
        uncover detailed information about the individuals
        and entities participating in the meeting. Your insights
        will lay the groundwork for strategic meeting preparation."""),
    verbose=True
  )
```

For example, the agent responsible for the `Research participants` task will have a background in research and will have a goal to conduct thorough research on the people and companies involved in the meeting. Feel free to tweak these parameters to see how they affect the agent's performance.

Tools

Tools are the resources that our agents will use to perform their tasks. These can be, for example, a search engine, a summarizer, a translator, etc.

Within CrewAI, tools are essentially wrappers around external APIs or services. They provide a way for our agents to interact with these services in a standardized way.

In this example, we are going to be using Exa as our search toolset. Exa is a powerful search engine that can be called through an API. Here is what a `Tool` object looks like in CrewAI:

```
from exa_py import Exa
from langchain.agents import tool

def search(query: str):
  """Search for a webpage based on the query."""
  return ExaSearchTool._exa().search(
      f"{query}", use_autoprompt=True, num_results=3
  )
```

Or, the full toolset:

```
import os
from exa_py import Exa
from langchain.agents import tool


class ExaSearchTool:
    @tool
    def search(query: str):
        """Search for a webpage based on the query."""
        return ExaSearchTool._exa().search(
            f"{query}", use_autoprompt=True, num_results=3
        )

    @tool
    def find_similar(url: str):
        """Search for webpages similar to a given URL.
        The url passed in should be a URL returned from `search`.
        """
        return ExaSearchTool._exa().find_similar(url, num_results=3)

    @tool
    def get_contents(ids: str):
        """Get the contents of a webpage.
        The ids must be passed in as a list, a list of ids returned from `se
        """

        print("ids from param:", ids)

        ids = eval(ids)
        print("eval ids:", ids)

        contents = str(ExaSearchTool._exa().get_contents(ids))
        print(contents)
        contents = contents.split("URL:")
        contents = [content[:1000] for content in contents]
        return "\n\n".join(contents)

    def tools():
        return [
            ExaSearchTool.search,
            ExaSearchTool.find_similar,
            ExaSearchTool.get_contents,
        ]

    def _exa():
        return Exa(api_key=os.environ["EXA_API_KEY"])
```

This is convenient because it exposes the method `tools()` that returns an array with all the tools that we have available. This way, we can easily pass this array to the agent when creating it.

This is what we did in the `research_agent` method above, where we passed the `ExaSearchTool.tools()` method to the `tools` parameter.

Processes

A process dictates the way that our agents will work together. In this case, we are using a sequential process, which means that each agent will work one after the other.

Another common process is the `hierarchical` process, where a manager (another agent) assigns tasks to other agents. This is useful when you have a more complex task that requires multiple sub-tasks to be completed. We will see this in action in another tutorial.

The process is defined when we create the crew. By default, the process is sequential. In order to set it to hierarchical, we would pass the `process` parameter as `Process.hierarchical`.

```
from crewai import Crew
from crewai.process import Process
from langchain_openai import ChatOpenAI

# Example: Creating a crew with a sequential process
crew = Crew(
    agents=my_agents,
    tasks=my_tasks,
    process=Process.sequential
)

# Example: Creating a crew with a hierarchical process
# Ensure to provide a manager_llm
crew = Crew(
    agents=my_agents,
    tasks=my_tasks,
    process=Process.hierarchical,
    manager_llm=ChatOpenAI(model="gpt-4")
)
```

Run your crew

Now that we have our crew set up, we can run it. This is done by calling the `kickoff` method on the crew object. This will start the process and the agents will start working on their tasks.

```
crew.kickoff()
```

This will return the final output of the crew. This output will be the result of the last task that was performed. In this case, it will be the summary and briefing of the meeting.

But note that we need to initialize all the previous objects in our code before we can run this. This means that we need to get the user input, create the agents, tasks and tools objects, and then create the crew object.

Here is an example of what the main file would look like:

```
from dotenv import load_dotenv
load_dotenv()

from crewai import Crew

from tasks import MeetingPreparationTasks
from agents import MeetingPreparationAgents

tasks = MeetingPreparationTasks()
agents = MeetingPreparationAgents()

print("## Welcome to the Meeting Prep Crew ##")
participants = input("What are the names of the participants in the meeting?
meeting_context = input("What is the context of the meeting?\n")
objective = input("What is your objective for this meeting?\n")

# Create Agents
researcher_agent = agents.research_agent()
industry_analyst_agent = agents.industry_analysis_agent()
meeting_strategy_agent = agents.meeting_strategy_agent()
summary_and_briefing_agent = agents.summary_and_briefing_agent()

# Create Tasks
research = tasks.research_task(researcher_agent, participants, meeting_conte
industry_analysis = tasks.industry_analysis_task(industry_analyst_agent, par
meeting_strategy = tasks.meeting_strategy_task(meeting_strategy_agent, conte
summary_and_briefing = tasks.summary_and_briefing_task(summary_and_briefing_

meeting_strategy.context = [research, industry_analysis]
summary_and_briefing.context = [research, industry_analysis, meeting_strateg

# Create Crew responsible for Copy
crew = Crew(
    agents=[
        researcher_agent,
        industry_analyst_agent,
        meeting_strategy_agent,
        summary_and_briefing_agent
    ],
    tasks=[
        research,
        industry_analysis,
        meeting_strategy,
        summary_and_briefing
    ]
)

result = crew.kickoff()


# Print results
print("## Here is the result")
print(result)
```

Delegating tasks

Awesome! Now that we have our tasks, agents, and tools defined. Remember that diagram up there? Well, that is an oversimplified version of how the afents actually work.

Remember that, when defining an agent, we assigned a set of tools to it. These tools will be passed in as a list in the prompt to the LLM. This is all happening behind the scenes, but the prompt will look something like this:

```
You are a {task.agent.role}. {task.agent.backstory}. Your goal is {task.agen
You have the following tools at your disposal: {task.agent.tools}.
Your task is to {task.description}.
```

Here is an example of what the prompt would look like for the `Research participants` task:

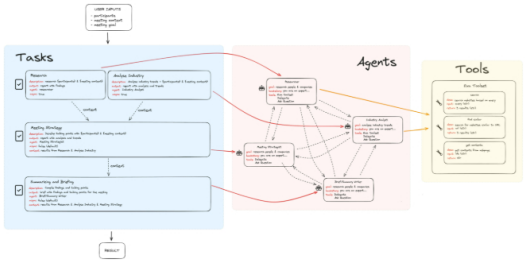
```
You are a Research Specialist. As a Research Specialist,
your mission is to uncover detailed information about the
individuals and entities participating in the meeting. Your
insights will lay the groundwork for strategic meeting preparation.
Your goal is to conduct thorough research on people and companies
involved in the meeting.

You have the following tools at your disposal:

- search: search(query: str) - Search for a webpage based on the query.
- find_similar: find_similar(url: str) - Search for webpages similar to a gi
- get_contents: get_contents(ids: str) - Get the contents of a webpage.
The ids must be passed in as a list, a list of ids returned from `search`.
- Delegate work to co-worker: Delegate work to co-worker(coworker: str, task
The input to this tool should be the coworker, the task you want them to do,
- Ask question to co-worker: Ask question to co-worker(coworker: str, questi
The input to this tool should be the coworker, the question you have for the

Your task is to Conduct comprehensive research on each of the
individuals and companies involved in the upcoming meeting.
Gather information on recent news, achievements, professional
background, and any relevant business activities.

Participants: {participants} Meeting Context: {meeting_context}.
```

Great. That gives us a good idea of what the agent will be doing. But take a closer look at the list of tools. You will see that there are two tools that we did not define in the `ExaSearchTool` class. These are the `Delegate work to co-worker` and `Ask question to co-worker` tools. What are they doing there?

Unless you disable these when initializing the agent, CrewAI will automatically add these tools to the agent. This means that agents can delegate tasks to other agents and ask questions to other agents depending on the context.

For example, if the briefing agent needs some extra information from the research agent, it can ask the research agent a question. Or, if the research agent is stuck and needs help from the industry analyst, it can delegate the task to the industry analyst.

So our nice diagram from above is not entirely accurate. Agents can actually communicate with each other, ask questions, and delegate tasks. This is what makes CrewAI so powerful. Our new diagram should look something like this:



As you can see, an AI Crew can be much more complex than just a list of tasks. Agents can communicate with each other, ask questions, and delegate tasks. And also, this is why we need to be very careful when designing our crew.

A note on costs

Remember that agents use a ReActive model, which means that they perform the chain-of-thought prompt in a loop until they think that they have completed the task. Every time they think about anything, they are making an API call to your LLM. This can be very expensive, if your agents are delegating tasks to each other all the time.

So, when designing your crew, make sure that you have a clear monitoring setup in place. You should be able to see how many API calls your agents are making, and how much it is costing you before using CrewAI in production.

A great tool for monitoring your API calls and everything that your agents are doing is LangSmith. It is a monitoring tool that allows you to see all the API calls that your agents are making, and how much it is costing you.

Conclusion

In this tutorial, we have introduced you to the basics of CrewAI and shown you how to create a simple crew using a sequential process. We have covered the concepts of tasks, agents, tools, and processes, and shown you how to plan your crew effectively.

I hope you have enjoyed it. Next tutorial will be a deeper dive into CrewAI. Stay tuned!

← How to use Streaming in LangChain and Streamlit
1 March 2024

How to automate Instagram Strategy with CrewAI →
5 April 2024

Privacy Policy Terms of Service